# Création de site immersif

## Préparation du TP

### Build

#### Webpack

Webpack est un **"module bundler"** open-source. Son objectif principal est de regrouper les fichiers JavaScript pour les utiliser dans un navigateur. Cet outil est également capable de transformer, regrouper ou empaqueter à peu près n'importe quelle ressource. Webpack est un outil conçu pour faciliter le développement et la gestion de **sites et d'applications web modernes**.

### Architecture

#### Express

Express.js est le framework standard pour le développement de serveur en Node.js. De plus, ce framework fonctionne très bien avec le CMS que nous allons utiliser.

#### Prismic

Prismic est le **Content Manager System (CMS)** que nous allons utiliser (nous n'allons quand même pas taper nos données en dur 😆).C'est plus précisément un **CMS Headless**, en effet il nous permet de choisir le framework, la technologie, la langage les plus adaptés pour notre projet, tout en fournissant une interface pour en gérer le contenu (contrairement à WordPress par exemple).

De plus, Prismic est un CMS assez intuitif, facile à utiliser pour les développeurs, et celui-ci a l'avantage de proposer une bonne documentation.

#### SEO

Une chose extrêmement importante à prendre en compte lors du développement d'un site web est l'optimisation pour les moteurs de recherche ou **SEO (Search Engine Optimization)**. Ce terme défini l'ensemble des techniques mises en oeuvre pour améliorer la position d'un site web sur les pages de résultats des moteurs de recherche (SERP). On l'appelle aussi référencement naturel (contrairement au SEA, référencement payant).

**On dit qu'un site est bien optimisé ou référencé si celui-ci se trouve dans les premières positions d'un moteur de recherche sur les requêtes souhaitées.**

Heureusement pour nous, la combinaison d'Express et de Prismic permet une bonne optimisation de notre site .

### Outils

#### Pug

Pug est un "template engine", un outil permettant d'écrire du code de manière différente, ici le code écrit différemment est le HTML.

Je vous invite à cliquer sur le lien ci-dessus pour vous familiariser avec la syntaxe de celui-ci. Cell-ci est épurée, élégante et propose plus de liberté au développeur.

#### SCSS

Sass est un langage de script préprocesseur compilé en CSS. Globalement, Sass contient des fonctionnalités inexistentes en CSS (mixins, variables, héritage, ...) et permet d'écrire beaucoup moins de CSS. Sass est disponible en deux syntaxes : Sass, qui ressemble à Pug, et SCSS, la syntaxe principale, qui est une amélioration de la syntaxe CSS3.

Dans notre cas, nous utiliserons SCSS.

*Note : vous allez enfin voir toute la puissance du CSS grâce à Sass*

#### Babel (ES6+)

Babel est un transcompilateur JavaScript. Mais qu'est-ce qu'un transcompilateur (et pourquoi un terme aussi compliqué ?). C'est un type de compilateur qui compile du code source d'un certain langage en du code source d'un autre langage. En ce qui concerne Babel, il permet de convertir du JavaScript en du ...JavaScript ! J'adore le concept, pas vous ?

Plus spécifiquement, il permet de convertir du code JavaScript récent (ECMAScript 2015+) en une version rétrocompatible de JavaScript pouvant être exécutée par des anciens navigateurs.

### Librairies

#### GSAP

GSAP ou GreenSock, *"the standard for modern animation"*, est un set d'outils JavaScript pour créer des animations. Tout ce que vous voyez sur un site web peut être animé avec GSAP. Que vous souhaitiez créer des animations d'UI élégantes ou des effets dynamiques dans des applications web, des jeux ou des histoires interactives, GSAP vous sera forcément utile.

GSAP offre la flexibilité et le contrôle dont nous avons besoin pour créer des site au niveau professionnel, mais aussi la facilité d'appréhension nécessaire pour les débutants. Quasiment tous les sites un minimum créatifs de notre époque utilisent GSAP.

*Note : Après avoir maîtrisé les bases, vous serez étonnés de tout ce que vous pouvez faire avec GreenSock*

#### Lodash

Globalement, Lodash est une librairie JavaScript moderne, principalement utilisée en back-end, qui permet de booster les performances JavaScript de votre site.

OGL est un framework minimaliste de WebGL partageant des fonctionnalités avec Three.js.

Cependant, je parlerais de cette librairie en détail plus tard.

## Installation

Il vous faut installer la dernière version de Node afin d'avoir accès au gestionnaire de paquet NPM dans votre terminal.

Puis rendez-vous à l'url ci-dessous et forkez/téléchargez le dépôt Git :

https://github.com/gabcaron/vuex

Tapez la commande suivante dans votre dépôt local pour installer toutes les dépendances :

```
npm install
```

#### Extensions VSCode recommandées

Pour mettre en place un environnement de travail optimal, je vous conseille les extensions suivantes : * **Bracket Pair Colorizer** (CoenraadS) * **EditorConfig for VS Code** (EditorConfig) * **ESLint** (Dirk Baeumer) : **must have** * **Import Cost** (Wix) * **Path Intellisense** (Christian Kohler) * **Shader languages support for VS Code** (slevesque)

L'extension ESLint va analyser votre code pour identifier les modèles problématiques trouvés dans le code JavaScript. Quand l'extension sera installée (je vous fourni le fichier *.eslint.js*), tapez *Ctrl + Shift + P* pour ouvrir le panneau de contrôle, tapez ensuite *ESLint : Fix* dans celui-ci et faites *Entrée* pour que ESLint corrige la syntaxe de votre code (pour que celui-ci soit à la norme ECMAScript 2015+).

### Configuration de la structure de votre projet avec Webpack

Comme vous pouvez le voir, je vous ai fourni les fichiers *webpack.config.build.js*, *webpack.config.js* et *webpack.config.development.js*, vous n'aurez pas besoin de créer votre configuration Webpack à la main. Je vous invite cependant à faire quelques recherches sur le fonctionnement et la mise en place de ceci.

N'oubliez pas de build votre projet et le lancer :

```
npm run build
```

Pour lancer votre projet par la suite, voici la commande :

```
npm start
```

Après cette commande (et après être arrivé à 100%), ouvrez votre localhost au port indiqué dans votre navigateur (je vous conseille Chrome pour le développement avec Node.js).

A la racine de votre projet, vous pouvez voir le fichier *package.json*. Celui-ci répertorie toutes les dépendances installés et utilisées dans votre projet, ainsi que des informations importantes : le nom du projet, sa version, sa description. Il contient aussi une partie scripts : celle-ci liste toutes les commandes que vous pouvez lancer en console concernant votre projet.

Exemple : `npm start` Cette commande tapée dans votre console équivaut à `concurrently --kill-others \"npm run backend:development\" \"npm run frontend:development\"` comme nous pouvons le voir. Cette commande tue les commandes si l'une meurt, run la commande créée `backend:development` qui correspond à `nodemon app.js` qui elle-même lance votre *app.js*, et enfin run `frontend:development` qui lance Webpack.

Cela peut paraître compliqué, il faut juste comprendre que nous avons créé un ensemble de petites commandes exécutables indépendamment, et nous avons regroupé une partie de ces commandes sous la commande *npm start*. Deux petits termes pour en représenter plus d'une dizaine, vous ne trouvez pas ça magique ?

### Garder ses dossiers et fichiers organisés

Tout développeur se doit de prendre cette habitude ! En effet, organiser ses dossiers et fichiers permet d'avoir du code plus lisible, de retrouver plus facilement des fichiers (composants, fichiers de style, vues, ...), cela facilite la compréhension du code et du projet, enfin votre code sera plus facile à utiliser !

Divisez votre code en petits fichiers.

Je vous ai fourni quelques dossiers, à vous de continuer dans cette voie .

*Note : pour mieux organiser votre projet, n'utilisez que les formats .woff et .woff2 pour vos polices importées*

## Structure du projet

### Structure des vues avec Express et Pug

Vous constatez l'existence d'un fichier *app.js* à la racine de votre projet. Ce fichier est la base même de votre projet, c'est dans celui-ci que nous allons mettre en place les routes de notre site.

Dans un premier temps nous allons créer les constantes nécessaires à l'utilisation d'Express et de notre serveur :

```
const path = require('path')
const express = require('express')
const app = express()
const port = 8004

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Ici, quand nous allons run notre projet, notre site sera accessible par l'URL **http://localhost:8004**.

Comme nous allons utiliser Pug, il nous faut le "dire" à notre projet :

```
// constantes

app.set('view engine', 'pug')
app.set('views', path.join(__dirname, 'views'))
app.locals.basedir = app.get('views')

// app.listen()
```

Ces quelques lignes indiquent à notre projet que le template engine utilisé ici est pug, et que les vues se situent dans le dossier 'views' à la racine du projet.

Nous allons ajouter les routes de nos pages :

```
// setup de pug

app.get('/', (req, res) => {
  res.render('pages/home')
})
```

Ce code fonctionne comme cela : quand notre application reçoit l'URL "**/**"(soit l'URL de base de notre site, sa racine), celle-ci doit afficher (render) la vue **home** (que nous allons créer juste après).

A vous d'ajouter les routes "/about", "/collections" et "/detail/:".

Tout ça est bien sympathique mais aucune vue n'est créée pour le moment ☹.

Créez dans votre dossier *views* un fichier *index.pug*. vous allez mettre ce code dans celui-ci :

```
extends /_includes/layout

block content
```

Ce fichier est la base de vos vues, et donc de ce qui est affiché sur votre site. Le block content correspond au block qui va stocker le contenu de vos vues.

Vous remarquez que ce fichier hérite du fichier *layout* dans le dossier _ *includes*. Créez celui-ci :

```
block variables
doctype html
html(class= isDesktop ? "desktop" : isTablet ? "tablet" : isPhone ? "phone" : "" lang="en")
  head
    include head
  body
    include preloader
    include navigation

    #content.content(data-template=template)
      block content

    include scripts
```

Le fichier *layout* va servir de plan pour vos vues. Il inclut *head* (la balise head en HTML), ainsi que la navigation et le preloader (ceux-ci seront présents sur toutes les pages, nous allons les gérer grâce au JavaScript). Nous pouvons voir que le contenu sera disposé dans une div appelée content (son attribut 'data-template' nous sera utile par la suite).

Continuons dans notre lancée. Pour le fichier *head* :

```
 meta(charset='utf-8')
block title
  title Titre
meta(name='viewport', content='width=device-width, initial-scale=1.0, maximum-scale=3.0')

<!-- Search Engine -->

meta(name="description" content="Site web immersif et moderne")

link(rel="apple-touch-icon" sizes="180x180" href="/apple-touch-icon.png")
link(rel="icon" type="image/png" sizes="32x32" href="/favicon-32x32.png")
link(rel="icon" type="image/png" sizes="16x16" href="/favicon-16x16.png")
link(rel="manifest" href="/site.webmanifest")
link(rel="mask-icon" href="/safari-pinned-tab.svg" color="#5bbad5")
meta(name="msapplication-TileColor" content="#da532c")
meta(name="theme-color" content="#ffffff")

<!-- Twitter -->

//-meta(name="twitter:card" content="summary_large_image")
//-meta(name="twitter:title" content=meta.data.title)
//-meta(name="twitter:description" content=meta.data.description)
//-meta(name="twitter:image" content=meta.data.image ? meta.data.image.url : '')

<!-- Open Graph general (Facebook, Pinterest & Google+) -->

//-meta(name="og:title" content=meta.data.title)
//-meta(name="og:description" content=meta.data.description)
//-meta(name="og:image" content=meta.data.image ? meta.data.image.url : '')
//-meta(name="og:type" content="website")

link(rel="stylesheet" type="text/css" href="/main.css")

base(href='/')
```

Pour *navigation* (le menu) :

```
nav.navigation
  .navigation__wrapper
    a.navigation__link(href='/')
      | Home

    ul.navigation__list
      li.navigation__list__item
        a.navigation__list__link(href="/about") About

      li.navigation__list__item
        a.navigation__list__link(href="/collections") Collections
```

Enfin, pour le preloader :

```
.preloader
    p.preloader__text
        | Preloader

        .preloader__number
            .preloader__number__text 0%
```

Créez aussi le fichier *script*, mais laissez le vide pour le moment.

Passons maintenant aux vues de nos pages. Dans le dossier *pages*, créez un fichier *home.pug* :

```
extends ../_includes/layout

block variables
  - var template = 'home'

block content
  h1.home__title Home
```

Nous avons ajouté ici un block de variables, et une variable nommée *template*. Cette variable détermine quelle vue afficher selon la route choisie.

Créez les vues *about*, *collections* et *detail*.

Maintenant, lancez votre projet grâce à `npm start`. Après le build, rendez-vous sur **http://localhost:8004**, vous devriez voir votre site !

Si vous changez l'URL pour accéder à la page '*/about*', vous devriez voir le contenu de votre site changer pour afficher le contenu de la vue *about*. De plus, si vous cliquez sur les liens du menu, votre site charge la page visée. Bravo, vous avez créé le routage de votre site ! Simple comme bonjour, non 😎?

### Intégrer Prismic à votre projet

Après avoir créé votre dépôt sur Prismic, vous allez compléter le fichier *.env* (cherchez les informations qu'il vous faut dans *Settings -> API Security* sur votre dépôt).

Maintenant que vous avez créé vos variables d'environnement, il faut les inclure dans votre *app.js*. Pour cela, il suffit d'ajouter cette ligne au début du fichier :

```
require('dotenv').config()
```

Pour indiquer que nous allons travailler avec Prismic, ajoutez ces lignes à votre liste de constantes :

```
const Prismic = require('@prismicio/client')
const PrismicDOM = require('prismic-dom')
```

Bien, commençons les choses sérieuses. Notre dépôt Prismic étant une API, il nous faut une fonction pour l'initialiser :

```
// Initialize the prismic.io api
const initApi = (req) => {
  return Prismic.client(process.env.PRISMIC_ENDPOINT, {
    accessToken: process.env.PRISMIC_ACCESS_TOKEN,
    req
  })
}
```

Où doit-on placer une telle fonction ? Je vous laisse y réfléchir...

Ensuite, nous aurons besoin d'une fonction capable de résoudre les liens de nos pages :

```
// Link Resolver
const HandleLinkResolver = (doc) => {
  // Define the url depending on the document type
  //   if (doc.type === 'page') {
  //     return '/page/' + doc.uid;
  //   } else if (doc.type === 'blog_post') {
  //     return '/blog/' + doc.uid;
  //   }

  // Default to homepage
  return '/'
}
```

Tout cela est bien beau, mais il nous manque un 'middleware' pour faciliter le transfert d'informations à venir :

```
// Middleware to inject prismic context
app.use((req, res, next) => {
  res.locals.Link = HandleLinkResolver
  res.locals.PrismicDOM = PrismicDOM
  res.locals.Numbers = (index) => {
    return index === 0
      ? 'One'
      : index === 1
      ? 'Two'
      : index === 2
      ? 'Three'
      : index === 3
      ? 'Four'
      : '';
  };

  next()
})
```

D'ailleurs, aviez-vous remarqué que la route utilisée par *app.get()* pour la page *detail* se finit par ':' ? Ajoutez-y '*uid*. L'identifiant unique (Unique Identifier ou UID) sera forcément utilisé dans l'URL *detail*, en fonction de la page détail que nous souhaitons afficher. Nous allons voir ça dans la suite.

### Implémenter le HTML sémantique avec Pug, Prismic et Express

*Question : que veut dire HTML sémantique et pourquoi l'utiliser ici ?*

Nous avons intégrer Prismic, mais nous ne l'utilisons toujours pas pour le moment... Il est temps de remédier à cela 😎!

Nous allons créer une fonction asynchrone *handleRequest*, qui sera appelée chaque fois que nous avons besoin de dialoguer avec l'API :

```
const handleRequest = async (api) => {
  const home = await api.getSingle('home')

  console.log(home)

  const assets = []

  home.data.gallery.forEach((item) => {
    assets.push(item.image.url)
  })

  console.log(assets)

  return {
    assets,
    home,
  }
}
```

Pour continuer cette fonction, vous devez comprendre plusieurs choses ici : * La fonction est asynchrone (*async*) pour ne pas bloquer le script JavaScript lors du chargement des données * Chaque appel à l'API se fait par un appel asynchrone lui aussi (*await*) * Le tableau *assets* contiendra les images de nos pages

Ensuite, modifiez la fonction de rendering de la page *home* :

```
app.get('/', async (req, res) => {
  const api = await initApi(req);
  const defaults = await handleRequest(api);

  res.render('pages/home', {
    ...defaults,
  });
});
```

Vous remarquez que l'ajout d'*async* : en effet, nous appelons les fonctions asynchrones *initApi* et *handleRequest*. Nous initialisons l'API Prismic, puis nous y faisons appel avec notre nouvelle fonction. Enfin, nous passons à notre vue les données récupérées grâce à *default* (d'ailleurs il me semble avoir déjà vu l'opérateur ... dans un TP précédent, saurez-vous me l'expliquer à nouveau ?).

Je vous laisse run votre projet et regarder dans votre terminal quand vous allez sur votre page *home*, comparez ce que vous y voyez à votre page Home sur Prismic.

Enfin, ajoutez ceci aux constantes :

```
// à ajouter sous path et express
const logger = require('morgan')
const errorHandler = require('errorhandler')
const bodyParser = require('body-parser')
const methodOverride = require('method-override')
```

Ajoutez aussi ceci juste au dessus de la fonction *initApi* :

```
app.use(logger('dev'))
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: false }))
app.use(errorHandler())
app.use(methodOverride())
app.use(express.static(path.join(__dirname, 'public')))
```

Je ne vais pas trop rentrer dans les détails, ce sont des outils qui nous seront utiles pour certaines choses spécifiques. Si vous voulez en apprendre plus, je vous invite à faire quelques recherches dessus.

Bien, à vous de jouer maintenant ! Récupérez les données de *meta*, *preloader*, *navigation*, *about* et *collections* (attention au dernier, la récupération de celui-ci est différent des autres). Pushez les images des pages dans le tableau assets (cherchez autour de *forEach()*), puis amusez-vous à console.log le tableau, vous verrez comment les images seront utilisées. Après ça, modifiez les fonctions de rendering des pages restantes (à vous de trouver comment render la page *detail*, cherchez autour de *GetByUID*). Modifiez la fonction *handleLinkResolver* en vous aidant des commentaires.

 Avant de passer à la suite, appelez-moi pour que je vérifie cette partie.

Il nous reste à modifier nos vues. Le contenu de l'API étant transmis à nos vues par les fonctions de rendering, voici comment nous allons l'utiliser. Remplacez le contenu du block *content* de la vue *home* par cela :

```
.home(data-background="#c97164" data-color="#f9f1e7")
  .home__wrapper
    .home__titles
      each collection, index in collections
        .home__titles__label=`${home.data.collection} ${Numbers(index)}`
        .home__titles__title=collection.data.title

    .home__gallery
      each media in home.data.gallery
        figure.home__gallery__media
          img.home__gallery__media__image(src=media.image.url alt=media.image.alt)

    a.home__link(href=Link(home.data.collections))=home.data.button
      svg.home__link__icon(xmlns="http://www.w3.org/2000/svg" viewBox="0 0 288 60")
        path(stroke="currentColor" fill="none" d="M456.5,98.5c-19.41,0-38.24.79-56,2.35a398.43,398.43,0,0,0-45.66,6.42c-6.49,1.35-
        path.home__link__icon__path(stroke="#FFC400" fill="none" d="M456.5,98.5c-19.41,0-38.24.79-56,2.35a398.43,398.43,0,0,0-45.6
```

Si vous avez bien suivi mes conseils, vous reconnaissez certainement le contenu de notre *console.log(home)* 😎. Je vous laisse remplacer certaines informations dans *head*, *navigation*, *preloader*, et je vous laisse intégrer le contenu des pages dans les vues *about*, *collections* et *detail* (utilisez quand vous en avez besoin les *each* - *in* et *if* de Pug, appelez-moi si besoin).

Je vous recommande fortement de reprendre le même principe de classes que celui que j'utilise ici, ça vous simplifiera énormément la tâche pour la prochaine partie !

 Avant de passer à la suite, appelez-moi pour que je vérifie cette partie.

## Intégrer le style avec Sass

Comme vous pouvez le voir dans le dossier *styles*, notre style sera divisé en plusieurs parties : * base : quelques règles de base (dont des règles de reset pour partir sur de bonnes bases) * components : nous mettons ici le preloader * layout : nous mettons ici ce qui touche au layout (tel que la navigation) * pages : chaque page sera représentée par un dossier ici, composé de fichier(s) * shared : les règles partagées entre différents éléments (titres, pages, ...) * utils : les variables, fonctions, ... * index.scss : le fichier de base, dans lequel nous allons faire appel aux fichiers Sass

Je vous ai fourni l'ensemble du style, je vous laisse tester quelques manipulations pour voir comment celui-ci fonctionne.

Avant de passer au JavaScript, vous devez : * Enlever le *display: none* du preloader.scss * Mettre les *img* en *opacity: 0* dans base.scss * Mettre *opacity: 0* et *visibility: hidden* dans pages.scss * Mettre *font-size: 0* au navigation__link dans navigation.scss

## JavaScript

Avant de commencer la programmation de cette partie (la plus intéressante j'imagine), place à un peu de culture générale.

La première apparition du JavaScript remonte à 1996 dans le navigateur Netscape. Face au succès de celui-ci, Microsoft sort une implémentation similaire dans son IE3.

Si JavaScript est aussi connu et utilisé, c'est grâce au web. En effet, tous les terminaux qui possèdent un navigateur sont capable d'interpréter ce langage.

Avec le succès du web, beaucoup de développeurs se sont ainsi mis au JavaScript par absence d'alternative, sans vraiment prendre la peine ni le temps de comprendre le langage, ce qui a donné des codes parfois illisibles et des pratiques pas très belles (jQuery ?). Tout ceci nous a mené à appeler le JavaScript le "Vilain petit canard" des langages de programmation.

Cependant depuis quelques années, l'ECMA (organisme chargé de rédiger les spécifications du JavaScript) ainsi que les éditeurs de navigateurs ont fait de gros efforts de standardisation. C'est d'ailleurs pour cela que je vous pousse à utiliser *ESLint* dans ce TP. A noter que grâce aux efforts fournis par bon nombre d'organismes, sans compter l'immense communauté autour du JavaScript, le langage est l'un des plus en vogue du moment, et l'un de ceux ayant le plus de potentiel.

En JavaScript, quasiment tout est Objet, les seules exceptions étant *string*, *number*, *boolean*, *null*, *undefined* et *symbol* qui sont des Primitives. Cette spécificité nous permet donc de créer chacun de ces types via son constructeur comme le reste du langage (pour pouvoir tout faire en objet). De plus, le langage est un **langage orienté objet à prototype** : un prototype étant un objet à partir duquel on crée de nouveaux objets.

Vous devez savoir que la partie qui suit ne fera qu'effleurer le langage, tout en vous apprenant à bien l'utiliser. Ce sera à vous, par la suite, de creuser le sujet et de faire des merveilles  !

### Mettre en place l'architecture JavaScript de notre site

Si vous regardez dans le dossier *app*, vous devriez y voir différents sous-dossiers : * *animations* : contiendra nos animations (title, paragraph, ...) * *classes* : nos classes seront disposées ici (page, button, ...) * *components* : contiendra les composants comme navigation ou preloader. Contient aussi nos canvas. * *pages* : chacune de nos pages aura son propre dossier contenant un index.js (même principe que le CSS) * *shaders* : ce dossier contient des shaders pour WebGL (je ne vais pas rentrer dans les détails sur cette partie, je vous laisse faire quelques recherches par rapport à ça) * *utils* : quelques éléments utilitaires (color, text)

En plus de ces dossiers, nous avons un *index.js*. C'est le fichier principal de notre application.

Pour commencer, dans l'index.js :

```
class App {
  console.log('App')
}

new App()
```

Normalement, vous devriez voir App apparaître dans votre console de navigateur.

Bien. Comme tout objet, notre *App* va avoir un constructeur dans lequel nous allons appeler nos méthodes. Créez celui-ci ainsi que la méthode createPages, qui instancie dans une map nos pages.

```
///
constructor () {
  this.createPages()
}

createPages () {
  this.pages = {
    about: new About(),
    collections: new Collections(),
    detail: new Detail(),
    home: new Home()
  }

  console.log(this.pages)
}
///
```

Pour que ce code fonctionne, nous devons créer nos objets de page. Dans *classes*, créez *Page.js* :

```
import GSAP from 'gsap'

import each from 'lodash/each'

export default class Page {
  constructor ({ element, elements, id }) {
    this.selector = element
    this.selectorChildren = {
      ...elements,
    }

    this.id = id
  }

  create () {
    this.element = document.querySelector(this.selector)
    this.elements = {}

    each(this.selectorChildren, (entry, key) => {
      if (
        entry instanceof window.HTMLElement ||
        entry instanceof window.NodeList ||
        Array.isArray(entry)
      ) {
        this.elements[key] = entry
      } else {
        this.elements[key] = document.querySelectorAll(entry)

        if (this.elements[key].length === 0) {
          this.elements[key] = null
        } else if (this.elements[key].length === 1) {
          this.elements[key] = document.querySelector(entry)
        }
      }
    })
  }

  show () {
    return new Promise((resolve) => {
      GSAP.from(this.element, {
        autoAlpha: 1,
        onComplete: resolve,
      })
    })
  }

  hide () {
    return new Promise((resolve) => {
      GSAP.from(this.element, {
        autoAlpa: 0,
        onComplete: resolve,
      })
    })
  }
}
```

Concrètement, chaque objet *Page* est composé d'elements (son contenu), d'une méthode *create()* qui crée la page (en lui intégrant son contenu), et de méthodes *show()* et *hide()* qui serviront à l'afficher et la faire disparaître (grâce à GreenSock).

Maintenant, nous devons créer nos objets de page. Dans *pages*, créez un dossier par page (About, Home, ...). Dans *About/index.js* :

```
import Page from 'classes/Page'

export default class About extends Page {
  constructor () {
    super({
      id: 'about',
      element: '.about',
      elements: {
        navigation: document.querySelector('.navigation'),
        title: '.about__title',
      }
    })
  }
}
```

Comme vous le constatez, nous donnons un id à notre page. De plus, le contenu de *element* correspond au contenu de l'élément de classe *.about*. Ce que nous intégrons dans *elements* sont des choses spécifiques, ici la navigation et le title de notre page.

A vous de créer les autres pages (attention, vous devrez override la méthode *create* pour *Home/index.js*, car celle-ci est la première page affichée).

Il ne nous manque plus qu'à importer ces pages fraichement créées dans notre *index.js*.

En plus de cela, nous allons créer une méthode _createContent() comme suit :

```
createContent () {
  this.content = document.querySelector('.content')
  this.template = this.content.getAttribute('data-template')
}
```

Créez une variable *page* dans *createPages()*, celle-ci doit être égale à la page de la map correspondant au template créé ci-dessus. Puis appelez sa méthode *create()* et sa méthode *show*.

Nous allons aussi créer deux nouvelles méthodes :

```
///
async onChange(url) {
  await this.page.hide()

  const res = await window.fetch(url)
  if (res.status === 200) {
    const html = await res.text()

    const div = document.createElement('div')
    div.innerHTML = html

    const divContent = div.querySelector('.content')
    this.content.innerHTML = divContent.innerHTML

    this.template = divContent.getAttribute('data-template')
    this.content.setAttribute('data-template', this.template)

    this.page = this.pages[this.template]
    this.page.create()
    this.page.show()
  } else {
    console.log(`response status: ${res.status}`)
  }
}

addLinkListeners () {
  const links = document.querySelectorAll('a')

  each(links, (link) => {
    link.onClick = (event) => {
      event.preventDefault()

      const { href } = link
      this.onChange(href)
    }
  })
}
```

Faites appel à ces méthodes dans le constructeur. Ces méthodes peuvent paraître compliquées, mais vous avez déjà vu leur principe avec Vue.js : le Single Page Application. Oui oui, nous n'avons besoin que de deux fonctions pour transformer notre site en Single Page Application ! La méthode *addLinkListeners()* s'occupe d'annuler le fonctionnement par défaut de nos liens et de faire appel à *onChange()* en lui passant le nouvel url, qui elle va changer le contenu de notre page grâce à la div de classe 'content' et à notre tableau de page en utilisant le template instancié plus haut.

Vous venez de comprendre en profondeur le fonctionnement d'un framework SPA, j'ajouterai même : vous venez de créer votre propre mini framework SPA, félicitations 😎!

### Structurer les composants et mettre en place des transitions de preloading

Nous allons créer notre composant de base. Dans *classes*, créez *Component.js* :

```
import EventEmitter from 'events'
import each from 'lodash/each'

export default class Component extends EventEmitter {
  constructor({ element, elements }) {
    super()

    this.selector = element
    this.selectorChildren = {
      ...elements,
    }

    this.create()

    this.addEventListeners()
  }

  create () {
    this.element = document.querySelector(this.selector)
    this.elements = {}

    each(this.selectorChildren, (entry, key) => {
      if (
        entry instanceof window.HTMLElement ||
        entry instanceof window.NodeList ||
        Array.isArray(entry)
      ) {
        this.elements[key] = entry
      } else {
        this.elements[key] = document.querySelectorAll(entry)

        if (this.elements[key].length === 0) {
          this.elements[key] = null
        } else if (this.elements[key].length === 1) {
          this.elements[key] = document.querySelector(entry)
        }
      }
    })
  }

  addEventListeners () {}

  removeEventListeners () {}
}
```

Vous remarquez les similitudes avec notre classe *Page*, je ne rentrerai donc pas dans les détails.

Dans notre dossier *components*, nous allons créer notre *Preloader.js* :

```
import GSAP from 'gsap'

import Component from 'classes/Component'

import each from 'lodash/each'

import { split } from 'utils/text'

export default class Preloader extends Component {
  constructor () {
    super({
      element: '.preloader',
      elements: {
        title: '.preloader__text',
        number: '.preloader__number',
        numberText: '.preloader__number__text',
        images: document.querySelectorAll('img')
```

```
      images: document.querySelectorAll('img'),
    },
  })

  split({
    element: this.elements.title,
    expression: '<br>',
  })

  split({
    element: this.elements.title,
    expression: '<br>'
  })

  this.elements.titleSpans = this.elements.title.querySelectorAll('span span')

  this.length = 0

  this.createLoader()
}

createLoader () {
  each(this.elements.images, (element) => {
    element.onload = (_) => this.onAssetLoaded(element)
    element.src = element.getAttribute('data-src')
  })
}

onAssetLoaded (image) {
  this.length++

  const percent = this.length / window.ASSETS.length

  this.elements.numberText.innerHTML = `${Math.round(percent * 100)}%`

  if (percent === 1) {
    this.onLoaded()
  }
}

onLoaded() {
  return new Promise((resolve) => {
    this.emit('completed')

    this.animateOut = GSAP.timeline({
      delay: 1,
    })

    this.animateOut.to(this.elements.titleSpans, {
      duration: 1.5,
      ease: 'expo.out',
      stagger: 0.1,
      y: '150%',
    })

    this.animateOut.to(
      this.elements.numberText,
      {
        duration: 1.5,
        ease: 'expo.out',
        stagger: 0.1,
        y: '100%',
      },
      '-=1.4'
    )
```

```
      this.animateOut.to(this.element, {
        autoAlpha: 0,
        duration: 1.5,
      })

      this.animateOut.call((_) => {
        this.destroy();
      })
    })
  }

  destroy() {
    this.element.parentNode.removeChild(this.element)
  }
}
```

Vous devez comprendre plusieurs choses ici : * Dans *elements*, *images* correspond aux images de notre site, que nous allons charger en amont (durant l'animation de preloading, d'où cette appellation) * *split* comme son nom l'indique va couper notre title pour permettre une meilleure animation d'apparition * *createLoader()* s'occupe de charger nos images en leur donnant leur url, ce qui facilite le chargement de notre site (et en améliore les performances, donc le référencement - Et oui tout est lié) * La méthode *onAssetLoaded()* incrémente le pourcentage vu lors du preloading (pour comprendre son fonctionnement, amusez-vous à console.log le pourcentage à chaque fois, vous verrez la vitesse de chargement de chaque image) * *OnLoaded()* anime la fin de chargement, en faisant disparaître le preloader * *destroy()* comme son nom l'indique détruit l'élément preloader de notre page, pour que celui-ci ne se lance pas à chaque changement d'url

Avant d'ajouter notre preloader à notre classe *App*, vous allez modifier les attributs *src* de chaque image de nos vues en *data-src*.

Ensuite, dans notre classe *App*, importez le Preloader. Créez une nouvelle méthode *createPreloader()* :

```
createPreloader () {
  this.preloader = new Preloader({})
  this.preloader.once('completed', this.onPreloaded.bind(this))
}
```

Enlevez *this.page.show()* de *createPages()*. Cet appel va maintenant trouver place dans une nouvelle méthode *onPreloaded()*, après avoir fait appel à la méthode *destroy()* de notre preloader. Enfin, appelez la méthode *addLinkListeners()* dans *onChange()*, après *show()*.

### Implémentation d'un smooth scroll et d'animations avec GSAP

Créer un site c'est bien, mais créer un site avec des animations sympas et propres c'est mieux, et c'est ce que l'on va faire 😎.

Pour cela, nous allons utiliser GreenSock. Je vous conseille fortement d'aller voir comment la librairie fonctionne avant de continuer le TP, car même si vous allez apprendre des choses basiques, GreenSock permet de faire énormément de choses.

Dans un premier temps, nous allons importer *Prefix* et *NormalizeWheel* dans notre classe *Page*, puis ajouter à notre constructeur cette ligne :

```
this.transformPrefix = Prefix('transform')
```

Dans *create()* :

```
// endroit où ce situe -> this.elements = {}

this.scroll = {
  current: 0,
  target: 0,
  last: 0,
  limit: 0,
}

this.onMouseWheelEvent = this.onMouseWheel.bind(this)
```

Vous comprennez ce que nous allons faire ?

C'est exact, nous allons gérer notre scroll avec GSAP, grâce à la récupération de l'événement de molette et aux positions du scroll !

Pour mettre tout cela en place, vous allez modifier les méthodes *show()* et *hide()* comme ceci :

```
show () {
  return new Promise((resolve) => {
    this.animationIn = GSAP.timeline()

    this.animationIn.fromTo(
      this.element,
      {
        autoAlpha: 0,
      },
      {
        autoAlpha: 1,
      }
    )

    this.animationIn.call((_) => {
      this.addEventListeners()

      resolve()
    })
  })
}

hide () {
  return new Promise((resolve) => {
    this.removeEventListeners()

    this.animationIn = GSAP.timeline()

    this.animationIn.to(this.element, {
      autoAlpha: 0
      onComplete: resolve,
    })
  })
}
```

Bien, maintenant que nous avons modifier ces deux méthodes pour qu'elles correspondent plus à nos besoins actuels, attaquons-nous à la gestion du scroll.

Nous allons créer trois méthodes : *onMouseWheel()*, qui va récupérer la position verticale (deltaY) et l'inclure dans scroll.target, *onResize()* qui va gérer la taille du client avec scroll.limit, et *update()*, qui va s'occuper du smooth scroll grâce à GSAP, à scroll.target et scroll.current :

```
onMouseWheel (e) {
  const { deltaY } = NormalizeWheel(e)
  this.scroll.target += deltaY
}

onResize () {
  if (this.elements.wrapper) {
    this.scroll.limit = this.elements.wrapper.clientHeight - window.innerHeight
  }
}

update () {
  this.scroll.target = GSAP.utils.clamp(
    0,
    this.scroll.limit,
    this.scroll.target
  )

  this.scroll.current = GSAP.utils.interpolate(
    this.scroll.current
    this.scroll.target,
    0.1
  )

  if (this.scroll.current < 0.01) {
    this.scroll.current = 0
  }

  if (this.elements.wrapper) {
    this.elements.wrapper.style [
      this.transformPrefix
    ] = `TranslateY(-${this.scroll.current}px`
  }
}
```

Essayez sans que scroll.limit ne soit la taille du client, vous comprendrez son importance capitale !

Je vous laisse créer *addEventListeners()* et *removeEventListeners()*, chacune faisant appel à sa fonction spécifique pour ajouter ou enlever (selon laquelle) l'événement 'mousewheel'.

Si vous avez bien fait attention, nous utilisons *elements.wrapper*, or celui-ci n'est pas présent dans nos pages. Ajoutez-le à *pages/About*.

Après avoir ajouté le scroll à notre classe *Page*, il ne nous reste qu'à l'ajouter à notre classe *App*.

Dans le constructeur, appelez *addEventListeners()* (attention aux emplacements) et *update()*.

Dans *onPreloaded()*, vous devez appeler *onResize()*, et vous devez l'appeler dans *onChange()* également.

Je vous laisse aussi créer les méthodes ci-dessus : * *onResize()* fait appel à *page.onResize()* si *page et _page.onResize()* existent * Idem pour update (concernant update), tout en créant *this.frame* et en l'implémentant (regardez *window.requestAnimationFrame*) * *addEventListeners()* ajouter l'événement 'resize'

### Animer les éléments

Créons tout d'abord une classe *Animation* :

```
import Component from 'classes/Component'

export default class Animation extends Component {
  constructor({ element, elements }) {
    super({ element, elements })

    this.createObserver()

    this.animateOut()
  }

  createObserver () {
    this.observer = new window.IntersectionObserver((entries) => {
      entries.forEach((entry) => {
        if (entry.isIntersecting) {
          this.animateIn()
        } else {
          this.animateOut()
        }
      })
    })

    this.observer.observe(this.element)
  }

  animateIn () {}

  animateOut () {}

  onResize () {}
}
```

Nous allons animer nos éléments *Highlight*, *Label*, *Paragraph* et *Title*, vous allez donc les créer dans le dossier *animations* :

```
import GSAP from 'gsap'
import Animation from 'classes/Animation'

export default class Highlight extends Animation {
  constructor({ element, elements }) {
    super({ element, elements })
  }

  animateIn() {
    GSAP.fromTo(
      this.element,
      {
        autoAlpha: 0,
        delay: 0.5
      },
      {
        autoAlpha: 1,
        duration: 1
      }
    )
  }

  animateOut() {
    GSAP.set(this.element, {
      autoAlpha: 0
    })
  }
}
```

A vous de créer les trois autres.

Créons aussi une classe *Colors* qui nous sera utile par la suite :

```
import GSAP from 'gsap'

class Colors {
  change({ backgroundColor, color }) {
    GSAP.to(document.documentElement, {
      background: backgroundColor,
      color,
      duration: 1.5
    })
  }
}

export const ColorsManager = new Colors()
```

Pour charger nos images de manière asynchrone, nous allons créer une classe *AsyncLoad* :

```
import Component from 'classes/Component'

export default class AsyncLoad extends Component {
  constructor({ element }) {
    super({ element })
    this.createObserver()
  }

  createObserver() {
    this.observer = new window.IntersectionObserver((entries) => {
      entries.forEach((entry) => {
        if (entry.isIntersecting) {
          if (!this.element.src) {
            this.element.src = this.element.getAttribute('data-src')
            this.element.onload = _ => {
                this.element.classList.add('loaded')
            }
          }
        }
      })
    })

    this.observer.observe(this.element)
  }
}
```

Et enfin une classe *Button* qui va nous servir pour la navigation principalement :

```
import GSAP from 'gsap'

import Component from 'classes/Component'

export default class Button extends Component {
  constructor ({ element }) {
    super({ element })

    this.path = element.querySelector('path:last-child')
    this.pathLength = this.path.getTotalLength()

    console.log(this.path.getTotalLength())

    this.timeline = GSAP.timeline({ paused: true })

    this.timeline.fromTo(
      this.path,
      {
        strokeDashoffset: this.pathLength,
        strokeDasharray: `${this.pathLength} ${this.pathLength}`
      },
      {
        strokeDashoffset: 0,
        strokeDasharray: `${this.pathLength} ${this.pathLength}`
      }
    )
  }

  onMouseEnter () {
    this.timeline.play()
  }

  onMouseLeave () {
    this.timeline.reverse()
  }

  addEventListeners () {
    this.onMouseEnterEvent = this.onMouseEnter.bind(this)
    this.onMouseLeaveEvent = this.onMouseLeave.bind(this)

    this.element.addEventListener('mouseenter', this.onMouseEnterEvent)
    this.element.addEventListener('mouseleave', this.onMouseLeaveEvent)
  }

  removeEventListeners () {
    this.element.removeEventListener('mouseenter', this.onMouseEnterEvent)
    this.element.removeEventListener('mouseleave', this.onMouseLeaveEvent)
  }
}
```

Voilà une bonne chose de faite !

Il ne nous reste plus qu'à créer notre composant *Navigation* :

```
import GSAP from 'gsap'

import Component from 'classes/Component'

import { COLOR_BRIGHT_GREY, COLOR_WHITE } from 'utils/color'

export default class Navigation extends Component {
  constructor ({ template }) {
    super({
      element: '.navigation',
      elements: {
        items: '.navigation__list__item',
        links: '.navigation__list__link'
      }
    })

    this.onChange(template)
  }

  onChange (template) {
    if (template === 'about') {
      GSAP.to(this.element, {
        color: COLOR_BRIGHT_GREY,
        duration: 1.5
      })

      GSAP.to(this.elements.items[0], {
        autoAlpha: 1,
        delay: 0.75,
        duration: 0.75
      })

      GSAP.to(this.elements.items[1], {
        autoAlpha: 0,
        duration: 0.75
      })
    } else {
      GSAP.to(this.element, {
        color: COLOR_WHITE,
        duration: 1.5
      })

      GSAP.to(this.elements.items[0], {
        autoAlpha: 0,
        duration: 0.75
      })

      GSAP.to(this.elements.items[1], {
        autoAlpha: 1,
        delay: 0.75,
        duration: 0.75
      })
    }
  }
}
```

Plus qu'à utiliser tout ça maintenant !

Tout d'abord, dans *Page*, importez *map* depuis lodash, puis nos animations, notre AsyncLoad et le ColorManager.

Dans le constructeur, ajoutez nos animations dans *selectorChildren* (syntaxe : animationHighlights), puis `preloaders: '[data-src]'` qui correspond à nos images.

Créez ensuite une méthode *createPreloaders()* qui implémente *this.preloaders* avec une map d'objets AsyncLoad formés d'*element*.

Maintenant, créez cette méthode :

```
createAnimations () {
  this.animations = []

  // Titles

  this.animationsTitles = map(this.elements.animationsTitles, (element) => {
    return new Title({
      element
    })
  })

  this.animations.push(...this.animationsTitles)

  // Paragraphs

  this.animationsParagraphs = map(
    this.elements.animationsParagraphs,
    (element) => {
      return new Paragraph({
        element
      })
    }
  )

  this.animations.push(...this.animationsParagraphs)

  // Labels

  this.animationsLabels = map(this.elements.animationsLabels, (element) => {
    return new Label({
      element
    })
  })

  this.animations.push(...this.animationsLabels)

  // Highlights

  this.aimationsHighlights = map(
    this.elements.aimationsHighlights,
    (element) => {
      return new Highlight({
        element
      })
    }
  )

  this.animations.push(...this.aimationsHighlights)
}
```

puis faites appel à ces deux méthodes à la fin de *create()*.

Il est temps de modifier *show()* :

```
  show (animation) {
    return new Promise((resolve) => {
      ColorsManager.change({
        backgroundColor: this.element.getAttribute('data-background'),
        color: this.element.getAttribute('data-color')
      })

      if (animation) {
        this.animationIn = animation
      } else {
        this.animationIn = GSAP.timeline()
        this.animationIn.fromTo(
          this.element,
          {
            autoAlpha: 0
          },
          {
            autoAlpha: 1
          }
        )
      }

      this.animationIn.call((_) => {
        this.addEventListeners()

        resolve()
      })
    })
  }
```

Enfin, ajoutez cette ligne à *onResize()* : `each(this.animations, (animation) => animation.onResize())`

Passons à notre *App* : importez la *Navigation*, puis faites appel à deux méthodes après *this.createContent()* : *createPreloader()* et *createNavigation()*.

Pour ce qui est de *createNavigation()*, cette méthode va instancier une navigation avec le template.

Pour finir, faites appel à la méthode *onChange()* de notre *this.navigation* dans *onChange()*, avec notre template en paramètre.

Pour paufiner notre travail, modifiez *Detail* et *Home*, le premier doit prendre son *button* dans *elements* et les deux doivent avoir *create()* qui fait appel au constructeur hérité en créant un nouveau link implémenté par le elements.button, et *destroy()* qui appelle la méthode *destroy* héritée et remove les événements des links créés par *create()*.

Notre site commence enfin à ressembler à quelque chose non 😀?

### API d'historique pour navigateur et UA Parser

Nous allons mettre en place la détection du device et du browser, cependant renseignez-vous sur ce qui a pu se passer concernant le UA Parser de Node (il faut faire très attention avec ce genre de librairie, celle-ci est un exemple concrêt de librairie visée par les hacks, en effet celle-ci récolte des données sur les device et browser utilisateurs).

Ajoutons une classe *Detection* :

```
class Detection {
  isPhone () {
    if (this.isPhoneChecked) {
      this.isPhoneChecked = true

      this.isPhoneChecked = document.documentElement.classList.contains('phone')
    }

    return this.isPhoneChecked
  }

  isDesktop () {
    if (this.DesktopChecked) {
      this.DesktopChecked = true

      this.DesktopChecked = document.documentElement.classList.contains('desktop')
    }

    return this.DesktopChecked
  }

  isTablet () {
    if (this.TabletChecked) {
      this.TabletChecked = true

      this.TabletChecked = document.documentElement.classList.contains('tablet')
    }

    return this.TabletChecked
  }
}

const DetectionManager = new Detection()

export default DetectionManager
```

Ajoutez à *App* :

```
   onPopState () {
    this.onChange({
      url: window.location.pathname,
      push: false
    })
  }
///
async onChange ({ url, push = true }) {
  ///
  if (res.status === 200) {
    ///
      if (push) {
        window.history.pushState({}, '', url)
      }
    ///
  }
}
///
addEventListeners () {
  window.addEventListener('popstate', this.onPopState.bind(this))
  ///
}
///
```

Pour finir avec la détection, passez `this.onChange(href)` en `this.onChange({ url: href })`

Vous avez maintenant la possibilité d'utiliser les flèches 'Précédent' et 'Suivant' du navigateur, sans bug au niveau de la page détail.